Simulation and Implementation of a Phase Vocoder on the TI C5502 DSP

Alex McAuley

EE 299, Winter 2013

Table of Contents

Introduction

This report outlines the design, simulation, and hardware implementation of a phase vocoder for the completion of UCLA's MSEE degree during the winter quarter of 2013. The phase vocoder is an algorithm capable of altering the pitch of an input sound in real-time, and as such can be used in musical performance applications such as auto-tuners. The phase vocoder algorithm has been known since 1966 [1][2], but its implementation has become more prevalent in recent years due to the availability of low-cost modern digital signal processors (DSPs). The implementation presented in this report uses Spectrum Digital's eZdsp5502 evaluation module for the Texas Instruments TMS320C5502 DSP as the hardware platform. Prior to discussing the hardware implementation and performance, the algorithm's theory of operation and MATLAB simulation results will be presented.

Background Theory

The phase vocoder is a means of achieving real-time pitch-shifting for audio signals, where pitch refers to the frequency content and characteristics of a signal. The claim that the algorithm works in real-time means that the algorithm works with only a minimal time-delay from input to output. The successful algorithm should operate quickly enough that the delay is imperceptible. The claim that the algorithm can shift a signal's pitch means that it should be able to shift the frequency content of the signal either up or down without distorting the shape of the overall spectrum beyond a simple stretching or compressing.

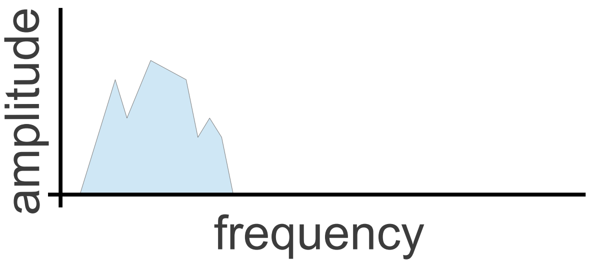

Figure 1a: Frequency spectrum of input signal.

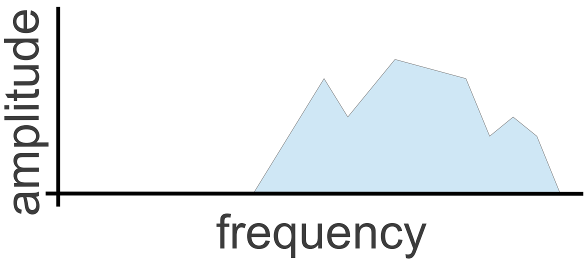

Figure 1b: Frequency spectrum of raised-pitch output.

For example, let Figure 1a represent the frequency spectrum of the input to a phase vocoder. Also let this particular phase vocoder be designed to raise the pitch of the input signal by one octave (a doubling of each frequency). Then the output spectrum would look that that of Figure 1b, where the spectrum is centered around a higher frequency and is more spread out, but the overall shape is unchanged. The successful phase vocoder must be able to produce this type of result even though it has no knowledge of what the complete signal will look like.

Offline Pitch Shifting

It is useful to first examine an offline, or non-real-time algorithm that can achieve the effect of Figure 1. By briefly exploring such an algorithm, a better sense of what the output signal should be can be obtained, as well as an improved respect for the complexity required to provide real-time performance. Figure 2 illustrates one such possible offline pitch-shifting algorithm (in this case, the pitch is raised). The input signal is digitized with an analog-to-digital converter (ADC), and the Fast Fourier Transform (FFT) is used to estimate the frequency content within discrete frequency "bins." Each frequency is then multiplied by a fixed pitch multiplication factor, and the original FFT content from that frequency is moved to the new frequency. Finally, the resulting output spectrum is turned into audible sound via the inverse FFT (IFFT) and a digital-to-analog converter (DAC).

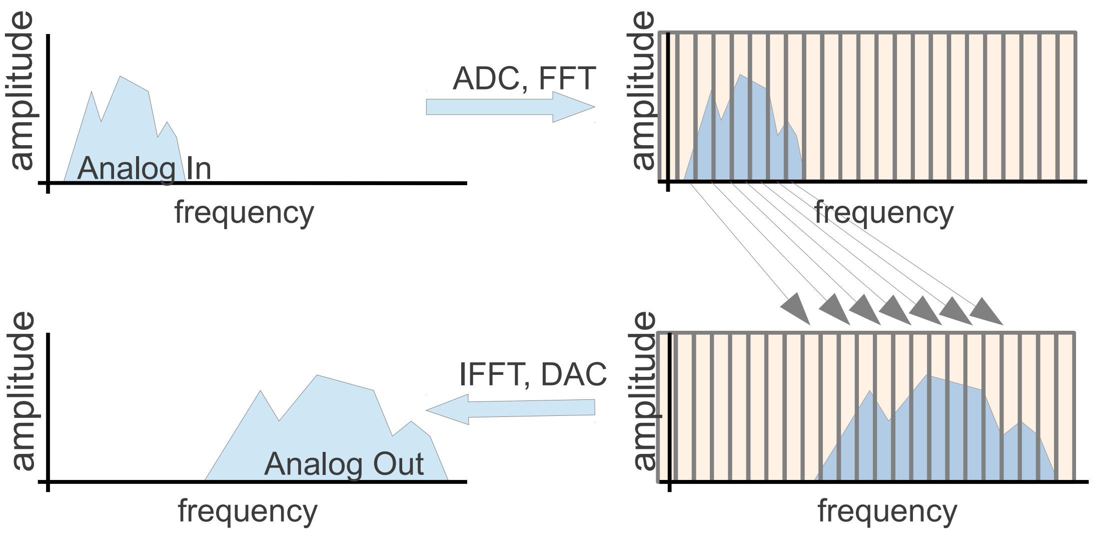

Figure 2:Offline algorithm for achieving pitch raising.

Although straightforward, the drawbacks of this algorithm make it very undesirable. The main problem with this algorithm is that taking the FFT of the entire signal naturally means that the entire signal must be obtained before any processing or playback can occur. While a modern computer would have plenty of memory available for signal storage, embedded devices for consumer applications might not. For example, a three minute song sampled at 44.1kHz (standard CD sampling rate) and 16-bit samples would require (180 sec) x (44100 samples/sec) x (16 bits/sample) = $127\text{x}10^6$ bits of memory, which is more than many embedded systems have available. Even worse than the memory requirements is the requirement to wait until the signal has been recorded before processing can begin. For the three minute song of the example, no output would be available until the entire song had played and finished processing (a minimum wait of three minutes). Such a wait is clearly undesirable.

Real-time Pitch Shifting

Because the phase vocoder works in real-time, it eliminates the long buffering period of the offline algorithm and simultaneously reduces the memory requirements. The remainder of this section focuses on describing how the algorithm works. Figure 3 , shown below, provides an example of the algorithm in pitch-raising mode, and this example will help illustrate the steps described in the following pages. Throughout this description, the variable names have been chosen to match nicely with those of [3].

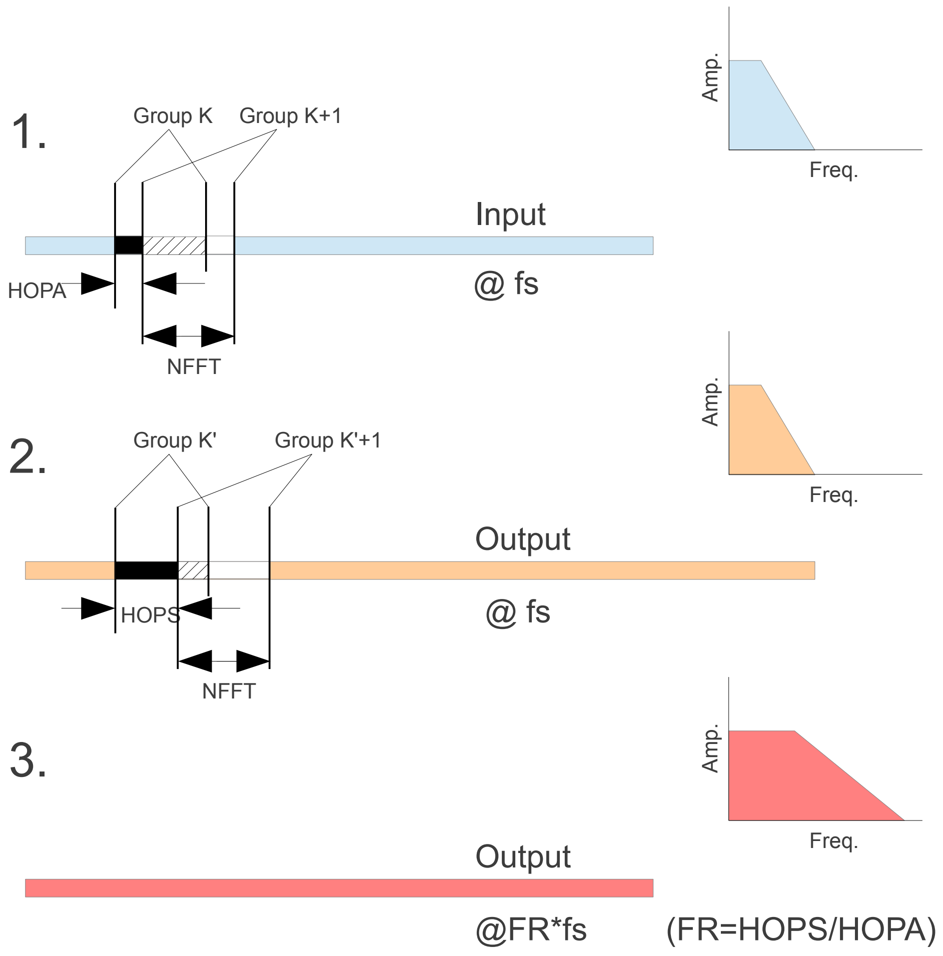

Figure 3:Phase vocoder pitch-raising.

Step 1 of Figure 3 represents an input signal (blue) that was sampled at a rate of *fs* samples per second being segmented into "groups" of *NFFT* samples, where *NFFT* is usually chosen to be a power of two. Each group overlaps the previous and next group by *NFFT-HOPA* samples, where *HOPA* is also commonly a power of two and is usually less than half the value of *NFFT*. The phase vocoder algorithm focuses on processing one group at a time, and so must process each group (of length *NFFT*) before the next group is ready for processing. This means that each group must be processed in less than *HOPA* x *fs* seconds in order to keep up with the flow of input samples.

In Step 2, groups are recombined with an inter-group spacing of *HOPS* (*HOPS* > *HOPA* in the case of Figure 3, which corresponds to a pitch-raising phase vocoder). If the sampling rate is unchanged from the input rate *fs*, the completed output signal will be longer (shorter if HOPS < HOPA) than the original signal by a factor of *FR* = *HOPS*/*HOPA*. The newly formed signal will still have the same frequency content as the original signal. This is because each group is just copied (along with some phase shifting as will be discussed soon) from the input to the output, and so no change in frequency content occurs on a block-by-block basis. The frequency content of each block is constant, and so, by extension, the input and output signals must contain the same frequencies. It is known that the frequency spacing of an FFT is given by 1/*T*, where *T* is the duration of the signal in seconds. In effect, then, Step 2 of the phase vocoder up-samples (down-samples) the frequency spectrum by a factor of *FR*, leaving the frequencies unchanged but lengthening (shortening) the signal in the time domain.

Step 3 uses a higher output sampling rate to transform the output signal from a longer-duration, identical-pitch version of the input[1] into an identical-duration, higher-pitch version. The output sampling rate is the input sampling rate multiplied by the duration-stretching factor, *FR*, from Step 2. Choosing this new sampling rate ensures that the output signal has the same duration as the original input while also multiplying each component frequency by the same factor, thus completing the pitch-raising. Note that while *FR* was greater than unity in this example and so corresponded to a pitch-raising algorithm, the same principles work just as easily with an *FR* less than or equal to unity. In fact, the final hardware implementation discussed later in this report demonstrates all three possibilities: *FR* values above, below, and equal to unity.

1 If stopped here (after Step 2), the algorithm could still have practical uses, *e.g.* dynamically altering signal duration to match an imposed, external timing. For example, instead of creating a pitch-correcting device, a tempo-correcting device could be used to keep a singer in a karaoke lounge in time with their background music.

Phase Adjustments

As mentioned in the previous section, Step 2 of the phase vocoder algorithm requires adjusting the phase of each group before adding it into the new output signal. If no phase adjustment were made to the output signal, large discontinuities would appear in the output, leading to severe audio distortion. Figure 4 demonstrates how this discontinuity occurs due to changing the spacing of the output groups.

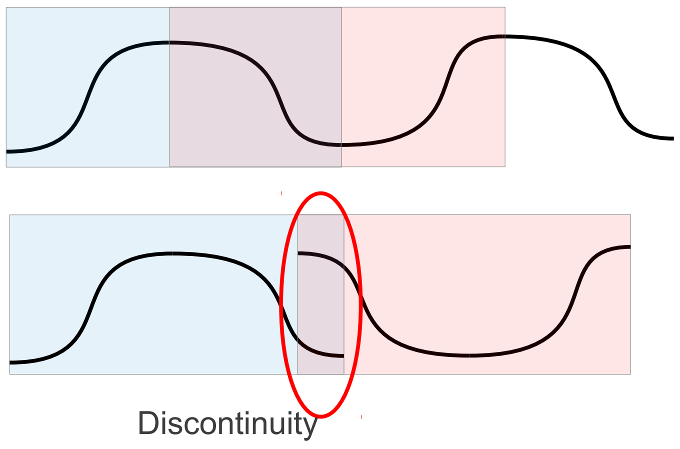

Figure 4:Distortion caused by new group spacing.

In the finished algorithm, each group has an *NFFT*-point Hanning window applied to it once the processing begins and again immediately before being added to the output signal, which greatly helps limit the effect of the distortion. Unfortunately, windowing alone cannot completely eliminate the distortion from these discontinuities. The discontinuity is caused by shifting the groups in time, and this time shift can equivalently be described as a frequency-dependent phase shift. Noting that frequency is the time-derivative of phase, this phase shift can be written as:

$$\Phi(rad)=\omega T=\hat{\omega} N \quad , \tag{1}$$

where *Φ* is the phase shift, *ω* is angular frequency, $\hat{\omega}$ is the angular frequency in the discrete-time representation of the group, and *N* is the number of samples corresponding to the time-domain shift. If the phase of the previous group is assumed to be known, the phase the new group should have to eliminate the discontinuity is found by adding the quantity $\hat{\omega}\times HOPS$ to the previous group's phase.

This is accomplished by taking the FFT of the input group, adding the phase correction factor for each frequency, and then taking the IFFT before re-spacing the groups. Although the phase of each group changes, its frequency content does not. Phase shifts are generally not audible to humans, and so the phase change is not perceptible by itself. The lack of discontinuity, however, is clearly perceptible as a reduction in noise as compared to a non-phase-corrected implementation.

Monitoring the phase shift of individual frequency components from group to group not only allows phase correction as just described, but it also provides a higher resolution estimate of which frequencies are present in the original signal. This in turn results in a more accurate output sound. This high-resolution frequency estimate is achieved by first rearranging Eq. 1:

$$\hat{\omega}=\frac{\Phi}{N} \quad . \qquad (2)$$

Eq. 2 states that any phase shift Φ over a fixed interval N can be expressed as a corresponding frequency deviation $\hat{\omega}$. When the FFT estimates the frequency spectrum as part of the phase-correcting, it does so by producing estimates of the spectrum in bins of width f*s*/*NFFT* Hz. Imagine for the sake of illustration that there is a bin of width 1Hz and centered at 10Hz. In this case the phase vocoder would not be able to distinguish between a 9.8Hz sine wave and a 10.2 Hz sine wave, since the bin resolution is not high enough. However, Eq. 2 shows how examining the phase shift in the 10Hz bin between successive group FFTs can yield a better frequency estimate: the difference between the bin center frequency and the true frequency is given the the phase difference (of the bin's center frequency) divided by NFFT. The phase difference used for this calculation is actually the phase difference of the two groups *minus the phase shift the bin's center frequency experienced just due to the group spacing*. That is, the frequency deviation from the bin's center frequency is calculated using only the phase shift that can't be explained away by assuming the center frequency accurately describes the signal in question.

One caveat when using Eqs. 1 and 2 is that the phase deviation between bins in successive groups must be limited to the range [0,2π], or equivalently [-π,π]. Values outside of this range can occur because the phase shift as computed in Eq. 1 grows linearly with the group spacing, but any amount outside of [0,2π] should "wrap" back into that range. If this "excess phase" were not removed, it would unduly affect future calculations. Figure 5 helps to illustrate this idea. The excess phase is readily removed using a modulo operation.

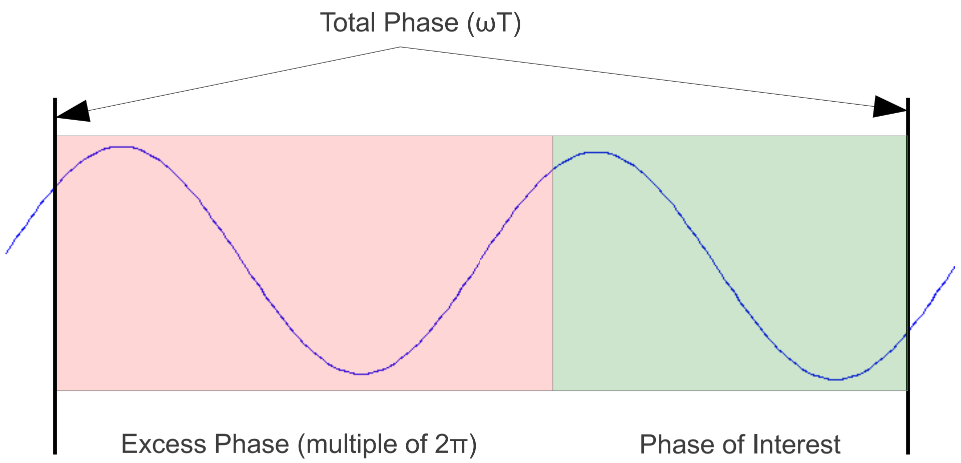

Figure 5:Illustration of excess phase arising from Eq. 1.

Phase-Correction Summary

The following steps summarize how the phase difference between two successive groups can be used to accurately measure frequency content in the original signal (Eq. 2), and how this accurate frequency estimate can in turn be used to create a more accurate phase-correcting factor (Eq. 1):

1. Calculate the phase difference in FFT bin j between group K and group $K\text{-}1$. Call this difference Δ_1 : $\Delta_1 = \angle(j_K) - \angle(j_{K-1})$.
2. Subtract from Δ_1 the phase due to the bin's center frequency having experienced *HOPA* samples after the previous bin: $\Delta_2 = \Delta_1 - \hat{\omega}_j \times HOPA$.
3. Eliminate the excess phase by wrapping the phase estimate to [-π,π]: $\Delta_3 = (\Delta_2 \bmod 2\pi) - \pi$.
4. Use Eq. 2 to calculate the corresponding deviation from the bin's center frequency: $\Delta\hat{\omega}_j = \dfrac{\Delta_3}{HOPA}$.

5. Form the improved-accuracy frequency estimate, $\underline{\hat{\omega}}_j$: $\underline{\hat{\omega}}_j = \hat{\omega}_j + \Delta\hat{\omega}_j$.

6. Calculate the phase shift of the group in the output signal by using the improved-accuracy frequency estimate and by adding this new phase to that of the previous group: $(New\ Phase)_j = (Old\ Phase)_j + \underline{\hat{\omega}}_j \times HOPS$.

Simulation

Three different MATLAB simulation files were written to test the algorithm prior to hardware implementation and to refine and study its performance once a working version was available. These scripts are available in the code listing under the names *pvshift.m*, *pvrt.m*, and *pvrtfull.m*. The first of these, *pvshift.m*, is an offline version of the phase vocoder algorithm. That is, it has the entire sound file available for processing at once. This version allowed the basic algorithm to be written and tested before moving on to a real-time simulation. Because it is only meant to be a first order test of the algorithm, filtering, quantization, and noise are all neglected. Similar code from [3] was consulted during the debugging process.

The second simulation script, *pvrt.m*, is a real-time version of the phase vocoder. It converts the code from *pvshift.m* into a form that works on input data one sample at a time, just like the hardware would. This function was used as a template for writing the first version of the DSP code.

The most advanced version of simulation code is found in *pvrtfull.m*. This incarnation of the algorithm includes noise, quantization, and input and output filtering. It was written in parallel with the final version of the DSP implementation so as to capture the actual hardware's behavior as well as possible.

Effects of HOPA and NFFT on Sound Quality

While writing the algorithm for the DSP, one question that arose was whether there was an optimal combination of *HOPA* and *NFFT* in terms of sound quality. In order to investigate this question, a metric for output signal distortion was needed. There are many such measures, but Total

Harmonic Distortion (THD) was chosen for it's relative simplicity. Under the THD metric, the distortion of a signal is measured as the ratio of a desired signal's power to the power contained in its harmonics. Harmonics are a good indicator of distortion in audio applications because they arise out of any nonlinearities in the system (e.g. clipping). The THD is computed as follows:

$$THD = \left(\frac{|A_f|^2}{|A_{2f}|^2 + |A_{3f}|^2 + |A_{4f}|^2 + \ldots} \right)^{-1} \quad ,$$

where A_k is the value of bin k in the FFT of the signal of interest, and A_f is the value in the bin corresponding to the frequency of the desired signal. For this test, one second of a 440Hz ("Concert A") sinusoid was chosen due it being neither especially high nor low in the audible spectrum. A test script, *pvrtfull_test.m* was created to vary the *FR*, *NFFT*, and *HOPA* of the *pvrtfull.m* simulation and measure the THD of the output. Table 1 shows the output of the tests for a fixed input noise 30dB below the signal level and with 16-bit quantization. Note that the values listed under the *HOPA* column are provided as fractions of *NFFT* (for example, ".25" means that *HOPA*=*NFFT*/4).

30db SNR, 16-bit Quantization

FR = .875				
HOPA\NFFT	**16**	**64**	**512**	**2048**
0.125	1.9783e+04	6.9764e-06	2.2859e-06	2.0494e-06
0.25	1.1513e+02	2.2993e-06	7.3359e-07	1.3388e-06
0.5	1.2526e-04	2.3579e-06	3.1090e-06	2.4374e-06
0.75	1.7206e-01	4.9153e-06	3.4609e-06	2.2206e-06

FR = 1				
HOPA\NFFT	**16**	**64**	**512**	**2048**
0.125	2.4928e-06	2.8551e-06	2.3124e-06	3.1397e-06
0.25	1.6080e-06	3.0059e-06	2.1029e-06	2.1337e-06
0.5	2.5382e-06	2.1085e-06	2.2623e-06	2.7136e-06
0.75	3.7692e-06	1.4091e-05	3.1729e-06	3.6471e-06

FR = 1.125				
HOPA\NFFT	**16**	**64**	**512**	**2048**
0.125	7.0017e+00	1.4485e-06	8.8469e-07	1.0173e-06
0.25	6.9285e+00	2.0565e-06	7.8466e-07	7.4530e-07
0.5	9.2209e-05	5.8510e-05	3.3716e-06	2.1533e-06
0.75	2.2433e-01	6.3345e-06	4.4879e-06	2.1637e-06

Table 1:Total Harmonic Distortion values at various NFFT, HOPA (as a fraction of NFFT), and FR.

One of the most interesting results of the simulation is that the distortion is fairly independent of parameter values for most situations. The *NFFT* value only causes much distortion when it is very low (16), and the *HOPA* value has minimal effect on distortion, at least in the range of tested values. Also of interest is the fact that the distortion is, for the most part, independent of whether the pitch is raised or lowered. What these results mean in terms of implementing the phase vocoder on the DSP is that parameter choice is not, at least in theory, critical so long as a *HOPA* of approximately 64 or greater is used.

Figure 6 shows a segment of simulated output in the time domain for *NFFT*=256, *HOPA*=64, and *FR*=1.125. The output is very clean, with only a small amount of noise visible. The slight difference in peak heights is attributed to the shape of the overlapping, time-offset Hanning window functions.

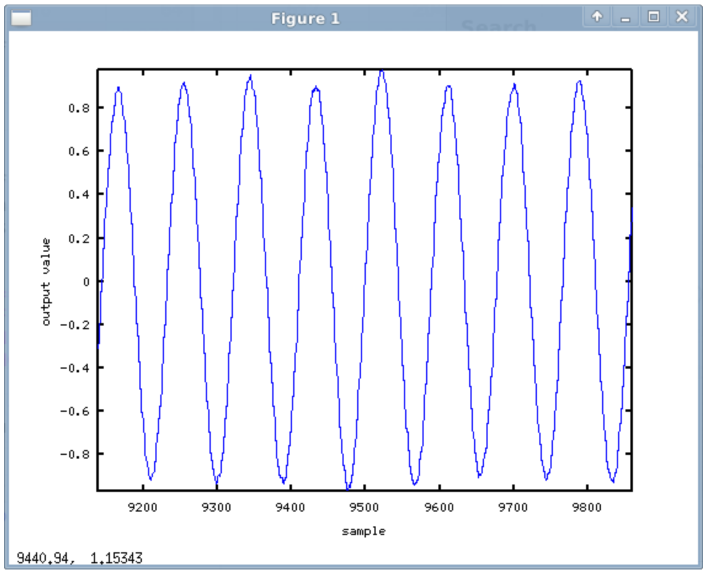

Figure 6:Full Simulation Output, NFFT=256, HOPA=64, FR=1.125.

Hardware Platform

Once the MATLAB simulation performed adequately, the algorithm was implemented on a DSP. Specifically, the Texas Instruments TMS320CC5502 DSP was chosen for its combination of performance, affordable price, and the existence of a company-provided signal processing library called DSPLIB that included FFT and IFFT functions. The processor itself is capable of running at over 200 million instructions per second and comes with supporting hardware as a part of an eZdsp Development Kit purchased from Spectrum Digital for $100. Most importantly, a 16-bit audio codec, the AIC3204, was included on this development board, and a sample C program for collecting and playing samples through it was included with the kit. This sample program formed the foundation for the hardware-implemented phase vocoder in that it provided working functions capable of collecting and writing sound data.

Whereas the original MATLAB simulations were conducted using floating-point numbers, the C5502 DSP was designed for 16-bit fixed-point numbers. Support for fixed-point calculations was therefore limited to software emulation, which provided the high precision and dynamic range of floating point calculations at the cost of high computational overhead. Most of the algorithm is carried out in fixed-point arithmetic so as to take advantage of the speed benefits.

In order to represent fractional numbers in this integer arithmetic, a two's-complement fixed-point number representation was used. For most of the calculations, the Q15 format was used, also known as 1.15. In this form of representation, the most significant bit of a 16-bit integer was used as a sign bit, and the remaining 15 bits represented the fractional portion. This system has a range of numbers from -1 to ($1-2^{-16}$), and it is the standard format used with the DSPLIB library functions.

The input and output filter coefficients, as well as the value of *FR*, can have values greater than one and so cannot use the Q15 format. By observing the fact that the highest value needed for these quantities is still less than 2, the 2.14 representation was chosen as a replacement in sections of the code that use those values. 2.14 is similar to Q15 (1.15) except that there are two "sign" bits and 14 fractional bits. The range of values representable in this notation is -2 to ($2-2^{-15}$).

Figure 7:C5502 eZdsp Development Kit. Photo: www.ti.com

Data Flow in the Hardware Implementation

Audio samples from the MP3 player are digitized and sent to the processor by the external codec. These samples are noisy; the codec itself produces approximately 2 LSBs of noise even when no input signal is present, and the cable connecting the development board to the MP3 player acts as an antenna that picks up 60 Hz noise from surrounding electronic devices. In order to reduce the high-frequency quantization noise in the output signal, a 2nd-order Butterworth lowpass filter with cutoff frequency at 6000 Hz filters all input samples. A Butterworth filter was selected due to its maximally flat passband characteristic, which minimizes frequency distortion. The chosen cutoff frequency eliminates much of the noise power while also maintaining much of the spectrum occupied by the input signals. The DC component of the input signal is also removed by subtracting the average of the previous group from the current group. The 60 Hz noise is slow compared to the group duration, and so is largely eliminated by this process.

Output from the initial filter is placed into a circular buffer of length 2 x *NFFT*. This length allows one half of the buffer to fill with new samples while the other half is processed by the rest of the algorithm. A circular buffer gets its name because the end of the buffer loops back around to its beginning; this structure easily allows writing and reading from different parts of the buffer simultaneously.

The *NFFT* samples corresponding the group undergoing processing are first run through the scaled FFT function provided in the DSPLIB. There is also an unscaled version available; the scaled version scales down the input to ensure no overflows occur during the FFT, whereas the unscaled version does not. The scaled version is used in this implementation because experiments with the unscaled version showed that it had a tendency to cause integer overflow on its output, which caused severe signal distortion.

The phase of each frequency in the FFT output is computed using the arctangent function provided by DSPLIB, and the result is subtracted from the phases of the previous group. As discussed earlier in the report, the phase due to the group spacing must also be subtracted. This group spacing phase is given by the product of the angular frequency and *HOPA*, which is a constant for each frequency. These constants are computed once when the device is powered on and stored in a lookup table for subsequent use, which greatly increases computational speed at the expense of memory usage. The development board contains 4Mb external flash memory, so there is plenty of memory available for lookup table storage.

The DSP stores the angular frequencies as normalized angular frequencies, meaning that the range [-π,π] is mapped to [-1,1], which is readily represented by Q15. One of the nice features about using Q15 for phase calculations is that every calculation is automatically computed modulo 2π. This is apparent if the possible two's complement values are represented as points an a circle with the bit pattern increasing clockwise, as demonstrated in Figure 8, where 0 would correspond to 0 radians on the unit circle, and -1 would correspond to -π. Just as the unit circle wraps around after π radians, so do two's complement numbers once -1 is reached.

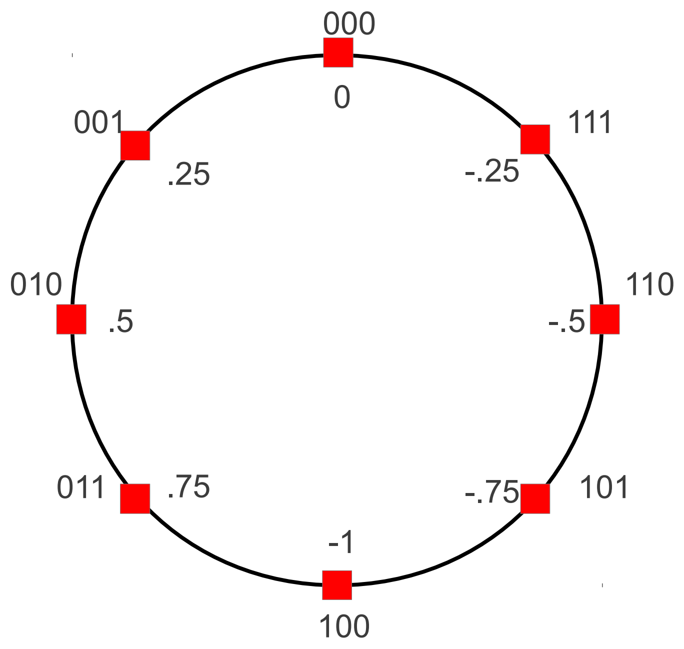

Figure 8:Two's complement 3-bit numbers represented on a unit circle.

Once the phases for the output group frequencies are calculated, the required phase shift must be applied to each frequency component. This is easy to express in polar coordinates:

$$A_{\text{out}}=|A_{\text{in}}|e^{j\Phi} \quad ,$$

where Aout and Ain are the input and output group frequency components for a particular frequency, and *Φ* represents the phase of the output. The processor, however, more readily works with Cartesian coordinates because it stores the real and imaginary parts of numbers in different arrays. The DSPLIB does not include any complex math functions, so the phase shift algorithm must be accomplished manually. Euler's identity easily simplifies the expression as follows:

$$A_{\text{out}}=|A_{\text{in}}|(\cos(\theta)+j\sin(\theta))$$
$$\operatorname{Re}(A_{\text{out}})=|A_{\text{in}}|\cos(\theta);\operatorname{Im}(A_{\text{out}})=|A_{\text{in}}|\sin(\theta)$$

The magnitude of each A_{in} is computed in the usual way according to the Pythogorean Theorem. These two calculations are carried out with floating-point arithmetic after using a DSPLIB function to convert Q15 to floating-point. This conversion is computationally intensive (and so is the subsequent reverse conversion), but it ensures a high degree of phase accuracy. The algorithm could be sped up by implementing a wholly fixed-point version of this calculation, but the slowdown in computation does not noticeably affect algorithm performance in this particular hardware setup, whereas less precise phase information might. The trade-off between accuracy and speed would need to be re-evaluated when attempting to implement this algorithm on a different processor.

The output of the IFFT for the current, recently processed group is windowed with a pre-computed Hanning window and stored in a two-dimensional output buffer that consists of the 5 most recently processed groups. Every time a new input sample arrives from the codec, a sum of the output samples from the overlapping groups is sent to the output lowpass filter and finally sent to the codec for audio output. The lowpass filter is the same as used on the input. By tying the output timing to the input timing, the algorithm ensures that there are no buffer under-run problems.

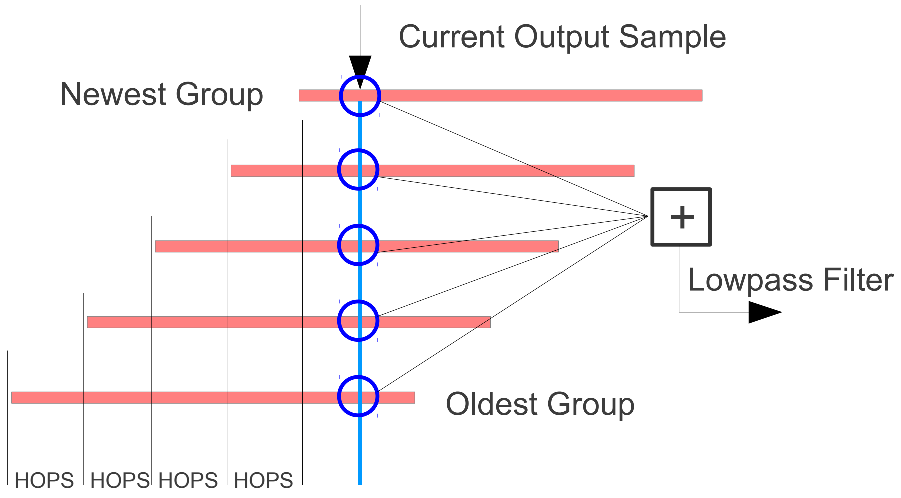

Figure 9:Adding samples from current and previous output groups in the delay buffer to form pre-lowpass audio output sample.

Results

Qualitatively, the hardware implementation of the phase vocoder works well. The unaltered audio sounds clean, and the raised-pitch and lowered-pitch algorithms are clearly distinguishable from each other and the unaltered input. In this sense, the hardware implementation is a success.

In order to gain a less subjective measure of performance, the algorithm was tested with a 440Hz input sinusoid (the same input as used in the simulation tests). Figure 10 Shows the clean input signal passed to the DSP from the audio codec and Figure 11 shows the similarly clean output when the hardware is not attempting to shift the pitch.

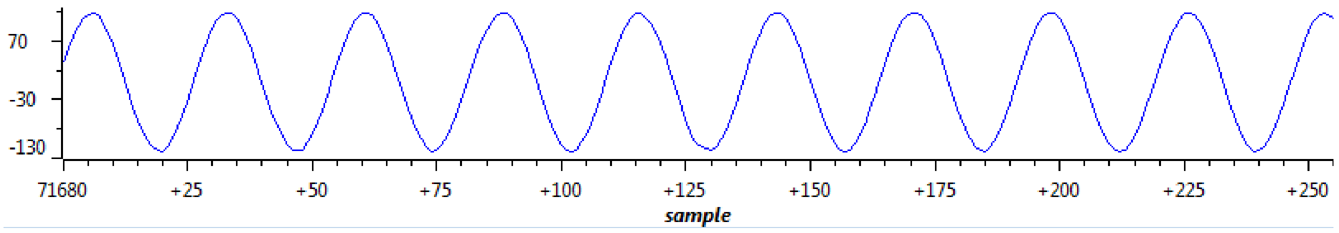

Figure 10: 440Hz input signal.

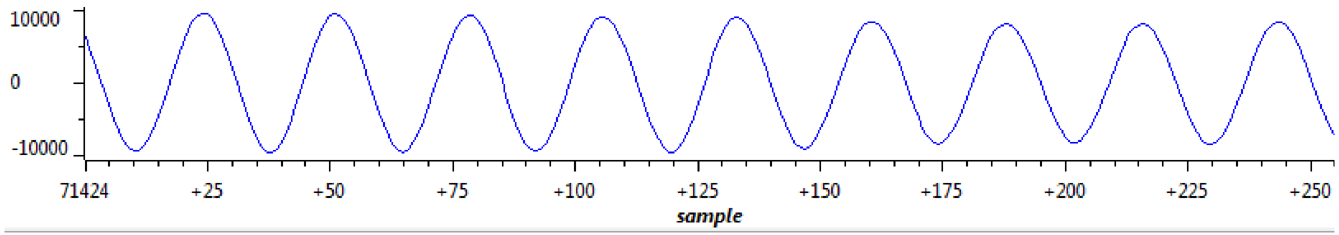

Figure 11: 440Hz output (no pitch change).

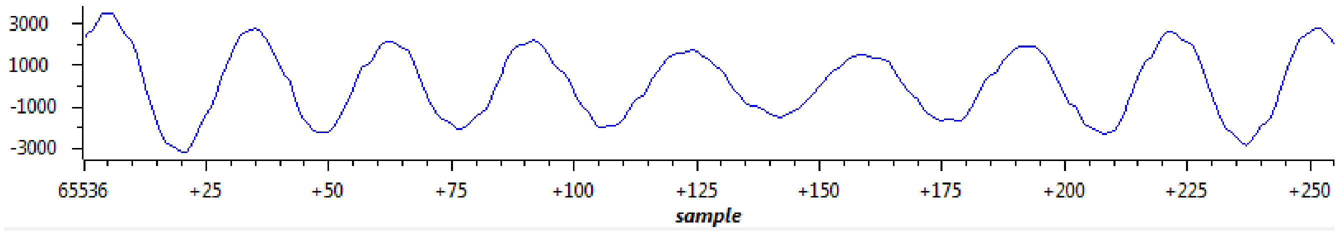

Figure 12: 385Hz output (lowered-pitch).

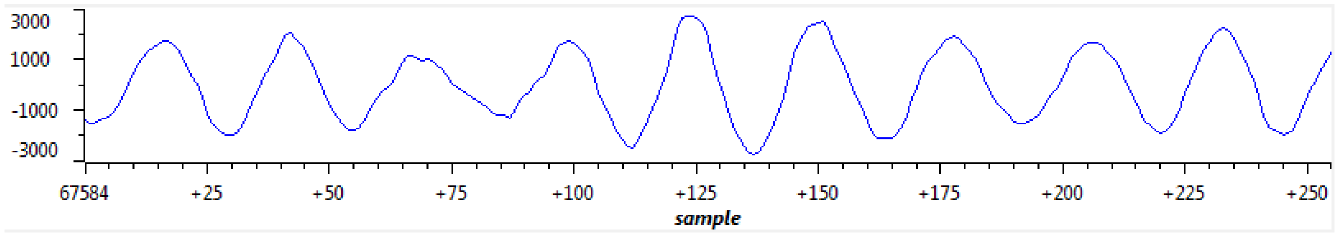

Figure 13: 495Hz output (raised-pitch).

Figure 12 and Figure 13 use the same 440Hz input signal, but the output pitches are shifted up and down by factors of 1.125 and 0.875, respectively. These outputs are clearly not as clean as the first two signals. Most noticeably, the amplitudes vary on a time-scale of a few periods. Despite extensive debugging efforts, the root cause of this amplitude variations remains unknown. The best hypothesis to date is that the variation is caused by an interaction of the HOPA and NFFT parameters and the choice of window function. This hypothesis is supported by simulation results that show differing degrees of amplitude variation depending on parameter choice. For example, Figure 6 from earlier in this report shows simulation output with a relatively low degree of amplitude variation. On the other hand, Figure 14 below shows a much higher degree of variation. Simulation results for the same settings as the hardware did not show as much variation as the actual hardware, but the fact that similar phenomena were reproducible in simulation indicates that a similar mechanism is at play.

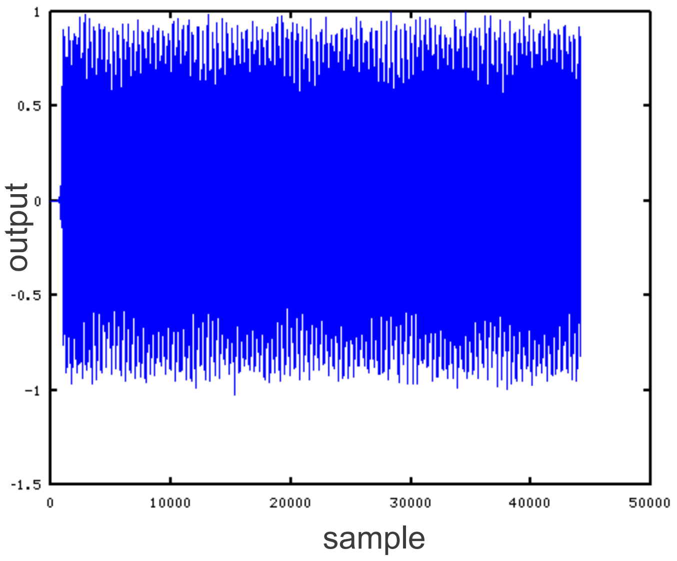

Figure 14: Output amplitude variation for 440Hz input, NFFT=1024, HOPA=32, FR=1.125

The effect of the amplitude modulation on output sound is not always noticeable, especially in spectrally busy signals that can drown out imperfections this modulation causes in the frequency spectrum. When it is noticeable, however, it sounds similar to a "chorus" effect, which occurs when a signal is superimposed with copies of itself that have been shifted very slightly in frequency. The amplitude modulation sounds this way because the modulation function can be expressed as $A+B \sin(2\pi f_m t)$, where $A < B$ and f_m is the modulation frequency. When mixed with the input sine wave, the A term yields a scaled version of the input, while the B sinusoid shifts the input frequency up and down by f_m. Although not always audible, the cause of the amplitude mixing would bear further investigation if this project were to be continued.

Conclusion

The MATLAB simulation of the phase vocoder was a success, and the hardware DSP implementation was largely successful as well. The output of the DSP implementation shows amplitude variation when a clean sinusoid is present at the input, but the algorithm still produces output that is clearly the pitch-shifted version of the input signal.

In terms of future work, eliminating the amplitude variation on the output should be the first priority. The second priority should be to work on optimizing the code, possibly at the assembly level, in order to ensure optimal performance (one option would be to re-write the code as a fully fixed-point implementation). Finally, a long term goal would be to develop a custom PCB around the C5502 DSP and a codec such as the currently-used AIC3204[2]. At that point, hardware power optimizations could be started and the device could be further developed, possibly to the point of becoming a product.

The phase vocoder is a powerful, real-time pitch-shifting algorithm that has been known for over four decades. Thanks to affordable, high-performance modern hardware such as the C5502 DSP, it is a very attractive option for pitch-shifting signals in consumer electronics as well as professional entertainment applications.

2 The AIC3204 datasheet is actually quite difficult to gain information from as there is little discussion of the many register values are important in typical applications and which can be safely ignored. Choosing a different codec might make software development easier in any future device.

Works Cited / Further Reading

[1] Dolson, Mark. "The Phase Vocoder: A Tutorial." *Jens Johansson - The Phase Vocoder: A Tutorial*. N.p., n.d. Web. 01 Dec. 2012. <http://www.panix.com/~jens/pvoc-dolson.par>.

[2] Flanagan, J. L., and R. M. Golden. "Phase Vocoder." *Bell System Technical Journal,* November (1966): 1493-509. Print.

[3] Grondin, François. "Guitar Pitch Shifter." *Guitar Pitch Shifter*. N.p., n.d. Web. 01 Dec. 2012. <http://guitarpitchshifter.com/>.

[4] Bernsee, Stephan. "Pitch Shifting Using The Fourier Transform." *The DSP Dimension RSS*. N.p., n.d. Web. 01 Jan. 2013. <http://www.dspdimension.com/admin/pitch-shifting-using-the-ft/>.

[5] Ellis, Dan. "A Phase Vocoder in Matlab." *A Phase Vocoder in Matlab*. N.p., n.d. Web. 10 Dec. 2012. <http://www.ee.columbia.edu/~dpwe/resources/matlab/pvoc/>.

Code Listing

MATLAB Simulation Code

pvshift.m

This code simulates the output of the phase vocoder, but does not model the real-time implementation. Code from [3] was consulted during the debugging of this function.

```
function [y,fso] = pvshift(x,fs,fr,nfft,hopa)
% OUTPUT y   = phase vocoder output
% OUTPUT fso = output sampling rate
% INPUT    x = input vector of sound samples
% INPUT   fs = sampling freq. (Hz) of input
% INPUT   fr = pitch multiplication factor
% INPUT nfft = FFT size (frame length)
% INPUT hopa = spacing between frames

hops = round(hopa*fr);

w = linspace(0,2*pi,nfft);
Nframe = floor((length(x)-nfft)/hopa);
y = zeros(1,(Nframe-1)*hops+nfft);
fso = round(fs*hops/hopa);
p = zeros(1,nfft);
hwin = hanning(nfft)';
%hwin=1;
fftbuf = zeros(1,2*nfft);
anglebuf = zeros(1,2*nfft);
newpntr = nfft+1;
oldpntr = 1;
temppntr = 0;
fftbuf(oldpntr:oldpntr+nfft-1) = fft(x(1:nfft).*hwin);
anglebuf(oldpntr:oldpntr+nfft-1) = angle(fftbuf(oldpntr:oldpntr+nfft-1));

for k=0:(Nframe-1)

   fftbuf(newpntr:newpntr+nfft-1) = fft(x((k+1)*hopa+1:(k+1)*hopa+nfft).*hwin);
   anglebuf(newpntr:newpntr+nfft-1) = angle(fftbuf(newpntr:newpntr+nfft-1));
   dphase = anglebuf(newpntr:newpntr+nfft-1)-anglebuf(oldpntr:oldpntr+nfft-1);
   dphase2 = dphase - hopa*w; %subtract off phase due to frame time spacing
   %wrap to -pi:pi range:
   dphasew = mod(dphase2+pi,2*pi)-pi;
   truew = w + dphasew/hopa;
   p = p + hops*truew; %cumulative output phase

   frameout = real(ifft(abs(fftbuf(newpntr:newpntr+nfft-1)).*exp(j*p))).*hwin;

   temppntr = newpntr;
   newpntr = oldpntr;
   oldpntr = temppntr;

   y(k*hops+1:k*hops+nfft) = ...
      y(k*hops+1:k*hops+nfft) + frameout;
endfor

end %end program
```

pvrt.m

This code models the phase vocoder in a real-time implementation, but does not model quantization or input noise effects.

```
function y = pvrt(x,fr,nfft,hopa)
% Simulation of real-time implementation of pvshift.
% This is much slower (in MATLAB, not on DSP) than pvshift.
% Use pvshift if there's no need to test accuracy of
% real-time phase vocoder implementation.
% OUTPUT   y = phase vocoder output
% INPUT    x = input vector of sound samples
% INPUT   fr = pitch multiplication factor
% INPUT nfft = FFT size (frame length)
% INPUT hopa = spacing between frames


%initialize misc. algorithm constants
lenx = length(x);
hops = floor(hopa*fr);
kmod = 0;
Noutb = 5; %number of output buffers
newoutbuf = Noutb-1; %index (minus 1) of newest output buffer

%stuff used in printing out conversion progress info
Nupdates = 1000;
kups = floor(linspace(1,lenx,Nupdates));
kn = 1;
kup = kups(kn);

%frequency vector
w = linspace(0,2*pi,nfft);

%generate templates for arrays
inbuf = zeros(1,hopa);
newbuf = zeros(1,nfft);
p = zeros(1,nfft);
fftbuf = zeros(1,2*nfft);
anglebuf = zeros(1,2*nfft);
outbufs = zeros(Noutb,nfft);
y = zeros(1,lenx);

%Hann window
hwin = hanning(nfft)';

%initialize pointers to frames within buffers
newpntr = nfft+1;
oldpntr = 1;
tempppntr = 0;

%algorithm - take one sample at a time and process it
for k = 1:lenx

  %!DO THIS PORTION DURING IRQ!%
  kmod = mod(k-1,hopa);
  inbuf(kmod+1) = x(k);

  out = 0;
  for m = 1:(Noutb-1)
  %start at 1 to simulate DSP filling newbuf while playing back older bufs
    bufindex = mod(newoutbuf-m,Noutb)+1;
    index = floor((kmod+hopa*(m-1))*fr);
      if((index ~= 0) && (index <= nfft))
        out = out + outbufs(bufindex,index);
      endif
   endfor

   y(k) = out;
   %!END OF IRQ PORTION%

   %!DO THIS PORTION ONCE EVERY TIME hopa SAMPLES ARE COLLECTED!%
   if(kmod == hopa-1) %new input buffer frame filled
      %Use circular buffer implementation on DSP for improved performance
      newbuf = [newbuf(hopa+1:end) inbuf];
      %

      fftbuf(newpntr:newpntr+nfft-1) = fft(newbuf.*hwin);
      anglebuf(newpntr:newpntr+nfft-1) = angle(fftbuf(newpntr:newpntr+nfft-1));
      dphase = anglebuf(newpntr:newpntr+nfft-1)-anglebuf(oldpntr:oldpntr+nfft-1);
      dphase2 = dphase - hopa*w; %subtract off phase due to frame time spacing
      %wrap to -pi:pi range:
      dphasew = mod(dphase2+pi,2*pi)-pi;
      truew = w + dphasew/hopa;
      p = p + hops*truew; %cumulative output phase

      frameout = real(ifft(abs(fftbuf(newpntr:newpntr+nfft-1)).*exp(j*p))).*hwin;

      temppntr = newpntr;
      newpntr = oldpntr;
      oldpntr = temppntr;

      %store newly computed out frame
      outbufs(newoutbuf+1,:) = frameout;
      newoutbuf = mod(newoutbuf+1,Noutb);

   endif
   %!END OF ONCE-EVERY-hopa-SAMPLES PORTION!%

   if(k==kups(kn))
      fprintf('%3.2f%% complete\n',100*k/lenx);
      fflush(1);
      kn = kn+1;
   endif
endfor

end %end program
```

pvrtfull.m

This is the most detailed of the simulation files, modeling some of the quantization and noise effects of the actual C5502 hardware implementation.

```
function y = pvrtfull(x,fr,nfft,hopa)
% Full simulation of real-time implementation of pvshift.
% Includes input quantization effects and input/output filtering
% Some implementation is slightly different than actual C5502
% code (eg circular vs linear buffers), but input/output relationship
% should be quite accurate.
% OUTPUT   y = phase vocoder output
% INPUT    x = input vector of sound samples
% INPUT   fr = pitch multiplication factor
% INPUT nfft = FFT size (frame length)
% INPUT hopa = spacing between frames
```

```
%simulation constants
NBITS_ADC = 16; %ADC depth
NBITS_PHASE = 16; %phase quantization depth
NOISE_PWR_DB = -30; %noise power relative to signal
[b,a] = butter(2,7/12);


%initialize misc. algorithm constants
lenx = length(x);
hops = floor(hopa*fr);
kmod = 0;
Noutb = 5; %number of output buffers
newoutbuf = Noutb-1; %index (minus 1) of newest output buffer

%stuff used in printing out conversion progress info
Nupdates = 10;
kups = floor(linspace(1,lenx,Nupdates));
kn = 1;
kup = kups(kn);

%frequency vector
w = linspace(0,2*pi,nfft);

%generate templates for arrays
inbuf = zeros(1,hopa);
newbuf = zeros(1,nfft);
p = zeros(1,nfft);
fftbuf = zeros(1,2*nfft);
anglebuf = zeros(1,2*nfft);
outbufs = zeros(Noutb,nfft);
y = zeros(1,lenx);

%Hann window
hwin = hanning(nfft)';

%add noise to input
x = awgn(x,-NOISE_PWR_DB);

%quantize input
x = quantize(x,[min(x),max(x)],NBITS_ADC);

%lowpass filter the input
x = filter(b,a,x);

%initialize pointers to frames within buffers
newpntr = nfft+1;
oldpntr = 1;
temppntr = 0;

%algorithm - take one sample at a time and process it
for k = 1:lenx

  %!DO THIS PORTION DURING IRQ!%
  kmod = mod(k-1,hopa);
  inbuf(kmod+1) = x(k);

  out = 0;
  for m = 1:(Noutb-1)
  %start at 1 to simulate DSP filling newbuf while playing back older bufs
    bufindex = mod(newoutbuf-m,Noutb)+1;
    index = floor((kmod+hopa*(m-1))*fr);
      if((index ~= 0) && (index <= nfft))
```

```
            out = out + outbufs(bufindex,index);
         endif
   endfor

   y(k) = out;
   %!END OF IRQ PORTION%

   %!DO THIS PORTION ONCE EVERY TIME hopa SAMPLES ARE COLLECTED!%
   if(kmod == hopa-1) %new input buffer frame filled
      %Use circular buffer implementation on DSP for improved performance
      newbuf = [newbuf(hopa+1:end) (inbuf-mean(newbuf(hopa+1:end)))];
      %

      fftbuf(newpntr:newpntr+nfft-1) = fft(newbuf.*hwin);
      anglebuf(newpntr:newpntr+nfft-1) = angle(fftbuf(newpntr:newpntr+nfft-1));
      dphase = anglebuf(newpntr:newpntr+nfft-1)-anglebuf(oldpntr:oldpntr+nfft-1);
      dphase2 = dphase - hopa*w; %subtract off phase due to frame time spacing
      %wrap to -pi:pi range:
      dphasew = mod(dphase2+pi,2*pi)-pi;
      quantize(dphase2,[-pi,pi],NBITS_PHASE);
      truew = w + dphasew/hopa;
      p = p + hops*truew; %cumulative output phase

      frameout = real(ifft(abs(fftbuf(newpntr:newpntr+nfft-1)).*exp(j*p))).*hwin;

      temppntr = newpntr;
      newpntr = oldpntr;
      oldpntr = temppntr;

      %store newly computed out frame
      outbufs(newoutbuf+1,:) = frameout;
      newoutbuf = mod(newoutbuf+1,Noutb);

   endif
   %!END OF ONCE-EVERY-hopa-SAMPLES PORTION!%

   if(k==kups(kn))
      fprintf('%3.2f%% complete\n',100*k/lenx);
      fflush(1);
      kn = kn+1;
   endif
endfor

   %filter the output
   y = filter(b,a,y);

   %normalize output vector
   mx = max(y);
   if(mx~=0)
      y = y./mx;
   endif

end %end program
```

pvrtfull_test.m

The following script automated the testing of parameter variation in the phase vocoder simulation.

```
%pvrtfull_test.m
%test output distortion of phase vocoder for various algorithm parameters

%input vector
```

```
tt = 0:1/44100:1-1/44100; %1 sec at audio rate
xx = sin(2*pi*440*tt); %440Hz test signal

%test nfft and hopa for different values of fr
nffts = [256];%[16,64,512,2048];
Nnffts = length(nffts);
hopas = [.25];%[.125,.25,.5,.75];
Nhopas = length(hopas);
frs = [1.125];%[.875,1,1.125];
Nfrs = length(frs);
fundamentals = frs*440;
%initialize output array
THDmat = zeros(Nhopas,Nnffts,Nfrs);
zz = Nhopas*Nnffts*Nfrs;
qq = 0;
for frindex = 1:Nfrs
    currentfr = frs(frindex);
    for nfftindex = 1:Nnffts
        currentnfft = nffts(nfftindex);
        for hopaindex = 1:Nhopas
            currenthopa = hopas(hopaindex);
            qq = qq + 1;
            %status update
            fprintf("============= %d%% OVERALL COMPLETETION =============\n",round(qq/zz*100));
            fprintf("%d of %d FRs\n",frindex,Nfrs);
            fprintf("%d of %d NFFTs\n",nfftindex,Nnffts);
            fprintf("%d of %d HOPAs\n",hopaindex,Nhopas);
            fprintf("------------------------------\n");
            %perform test with current parameter values
            yy = pvrtfull(xx,currentfr,currentnfft,currenthopa*currentnfft);
            yfft = fft(yy);
            %calc THD
            fundpwr = abs(yfft(fundamentals(frindex)+1))^2;
            %harmonic distortion pwr (look at first 25 harmonics)
            distpwr = 0;
            for(k = 2:26)
                distpwr = distpwr + abs(yfft(k*fundamentals(frindex)+1))^2;
            end
            THD = distpwr/fundpwr;
            %store result
            THDmat(hopaindex,nfftindex,frindex) = THD;
            fprintf("*** THD = %f ***\n",THD);
        endfor
    endfor
endfor
```

quantize.m

The quantize function was used to simulate quantization effects.

```
function y = quantize(x, range, N)
% quantize (signed) signal x to N levels based on input
% range (range(1),range(2)), return quantized version in
% signal y - values in range (0,1)
   a = x-range(1);
   d = a./(range(2)-range(1));
   y = range(1) + round(d*(2^N-1))/(2^N-1)*(range(2)-range(1));
end
```

Code for the C5502

The following functions represent the custom files used to create the phase vocoder on the

C5502 processor and development board. Due to space limitations, code that came with the development board or manufacturer-provided functions that were used as-is are not reproduced here. All code was developed under TI's Code Composer Studio ver. 4.2.4.00033 on a Windows 7 PC, and use was also made of TI's DSPLIB ver. 2.40.00 (http://www.ti.com/tool/sprc100) and Spectrum Digital's board support library (http://support.spectrumdigital.com/boards/ezdsp5502/revc/files/ezdsp5502_BSL_RevC.zip) .

Code for the phase vocoder was based on the AIC3204 sample project included with purchase of the development board.

main.c

The starting point of the phase vocoder code.

```
#include "stdio.h"
#include "ezdsp5502.h"
#include "ezdsp5502_i2cgpio.h"
#include "csl_gpio.h"
#include "pvplaydefs.h"

extern Int16 aic3204_IO_loop(void);
extern Int16 readswitches(Int16 *switches);

extern Int16 switches[2];
extern Int16 Px;
/*
 *
 *  main( )
 *
 */
void main( void )
{
   /* Initialize BSL */
   EZDSP5502_init( );

   /* Setup I2C GPIOs for Switches */
   EZDSP5502_I2CGPIO_configLine(  SW0, IN );
   EZDSP5502_I2CGPIO_configLine(  SW1, IN );
   EZDSP5502_I2CGPIO_configLine(  LED0, OUT);
   EZDSP5502_I2CGPIO_configLine(  LED1, OUT);
        printf("start...\n");
        EZDSP5502_I2CGPIO_writeLine(LED0,0);
        EZDSP5502_I2CGPIO_writeLine(LED1,0);
        readswitches(switches);
        while(switches[0]){
                readswitches(switches);
                if(switches[1]){
                        Px++;
                        if(Px==NPHASE){
                                Px=0;
                        }
                        EZDSP5502_I2CGPIO_writeLine(LED0,Px==2);
                        EZDSP5502_I2CGPIO_writeLine(LED1,Px==0);
                        while(switches[1]){
                                readswitches(switches);
                        }
                }
        }

   aic3204_IO_loop( );
}
```

aic3204_IO_loop.c

This is the non-interrupt portion of the code wherein the groups are processed.

```
#include "stdio.h"
#include "ezdsp5502.h"
#include "ezdsp5502_mcbsp.h"
#include "csl_mcbsp.h"
#include "ezdsp5502_i2cgpio.h"
#include "csl_gpio.h"
#include "math.h"
#include "dsplib.h"
#include "pvplaydefs.h"

/*prototypes*/
void config_AIC3204();
Int16 build_wHatVecs(DATA *wHat,float *fwHat,DATA *hopaXwHat);
Int16 build_Windows(DATA *hanning, DATA *raisedhanning);
Int16 readswitches(Int16 *switches);
extern Int16 AIC3204_rset( Uint16 regnum, Uint16 regval);

/*Input FFT*/
#pragma DATA_SECTION (x,".input")
DATA x[2*NFFT];

/*Output FFT (Q15 and float versions)*/
#pragma DATA_SECTION (y,".input")
DATA y[2*NFFT];
#pragma DATA_SECTION (fy,".f1")
float fy[2*NFFT];
DATA yr[NFFT];
DATA yi[NFFT];

/*Phase info*/
#pragma DATA_SECTION (phase,".miscCE0")
DATA phase[2*NFFT];                   //contains new and old data
#pragma DATA_SECTION (dphase,".miscCE0")
DATA dphase[NFFT];                    //new phase - old phase
//#pragma DATA_SECTION (fdphase,".f3")
#pragma DATA_SECTION (fdphase,".miscCE0")
float fdphase[NFFT];
float ftotalPhase[NFFT];
Int16 newphasepntr=0;

/*Magnitude of freq. components*/
float fmags[NFFT];

/*frequency vectors*/
#pragma DATA_SECTION (fwHat,".f2")
float fwHat[NFFT];
DATA wHat[NFFT];
DATA hopaXwHat[NFFT];
float ftruewHat[NFFT];

/*Windows*/
DATA hanning[NFFT];
DATA raisedhanning[NFFT];

Int16 inbuf[2*NFFT];                  //new samples buffer
Int16 outbuf[NOUT][NFFT];//output samples buffer
```

```
Int16 bufindex = 0;                //index into in/out bufs
Int16 bindex=0;
Int16 outbufindex=0;

/*switch state*/
Int16 switches[2]={0,0};
Int16 oldswitch0=0;

/*Phase shift*/
Int16 Px=1;
Int16 hops_[NPHASE] = HOPS;
Int16 fr_[NPHASE] = FR;

/*misc.*/
volatile Int16 i=0;          //loop counter
Int16 temp;
Int16 temp2;
long cntr=0;

//test
Int16 lqq = 10;
Int16 bufstore[10];
Int16 qq=0;


/*
 *
 *  AIC3204 IO Loop
 *      Loops audio from LINE IN to LINE OUT, performs intermediate processing
 *
 */
Int16 aic3204_IO_loop( )
{

        config_AIC3204();

        printf("starting EZDSP5502_MCBSP_init( )\n");
  /* Initialize McBSP & IRQ*/
  EZDSP5502_MCBSP_init( );

        /*build required constant buffers*/
        build_wHatVecs(wHat,fwHat,hopaXwHat);
        build_Windows(hanning,raisedhanning);

        printf("starting playback (cfft edition)...\n");
  while(1){
        if(bufindex==0 || bufindex==HOPA || bufindex==2*HOPA
                || bufindex==3*HOPA || bufindex==4*HOPA || bufindex==5*HOPA
                || bufindex==6*HOPA || bufindex==7*HOPA){

                        temp = bufindex;
                        outbufindex = outbufindex+1;
                        outbufindex = outbufindex==NOUT?0:outbufindex;

                if(temp==0){
                        bindex=4*HOPA;
                }
                else if(temp==HOPA){
                        bindex=5*HOPA;
                }
                else if(temp==2*HOPA){
                        bindex=6*HOPA;
                }
                else if(temp==3*HOPA){
                        bindex=7*HOPA;
                }
                else if(temp==4*HOPA){
                        bindex=0;
                }
                else if(temp==5*HOPA){
                        bindex=HOPA;
                }
                else if(temp==6*HOPA){
                        bindex=2*HOPA;
                }
                else if(temp==7*HOPA){
                        bindex=3*HOPA;
                }


                /*Fill FFT input vector*/
                temp = bindex;
                for(i=0;i<NFFT;i++){
                        x[2*i]=(Int16)(((long)inbuf[temp]*hanning[i])>>15);
                        x[2*i+1]=0;
                        y[2*i+1]=0;
                        temp++;
                        if(temp==2*NFFT){
                                    temp=0;
                        }
                }

                /*FFT*/
                cfft(x,NFFT,NOSCALE);
                cbrev(x,y,NFFT);

                /*calculate phase info*/
                for (i=0;i<NFFT;i++){
                        /*new phase*/
                        yr[i]=y[2*i];
                        yi[i]=y[2*i+1];
                }
                atan2_16(yr,yi,&phase[newphasepntr+i],NFFT);
                for(i=0;i<NFFT;i++){
                        dphase[i] = phase[newphasepntr+i]-phase[NFFT-newphasepntr+i]-hopaXwHat[i];
                }
                newphasepntr = NFFT-newphasepntr;
                q15tofl(dphase,fdphase,NFFT);
                q15tofl(y,fy,2*NFFT);
                /*now dphase is in the normalized freq. range [-1.0f, 1.0f]*/
                for(i=0;i<NFFT;i++){;
                        //can perform using bit operations if compiler doesn't automatically
                        ftruewHat[i] = fwHat[i]+fdphase[i]/HOPA;
                        //same note as above
                        ftotalPhase[i] = (fmod((ftotalPhase[i]+ftruewHat[i]*hops_[Px]+1.0),2.0)-1.0)*3.14159f;
                        /*calculate magnitudes*/
                        fmags[i] = sqrt(fy[2*i]*fy[2*i] + fy[2*i+1]*fy[2*i+1]);
                        /*real*/
                        fy[2*i] = fmags[i]*cos(ftotalPhase[i]);
                        /*im*/
                        fy[2*i+1] = fmags[i]*sin(ftotalPhase[i]);
                }

                fltoq15(fy,y,2*NFFT);
```

```
                        /*Inverse FFT*/
                        cifft(y,NFFT,NOSCALE);
                        cbrev(y,y,NFFT);

                                /*store output*/
                                for(i=0;i<NFFT;i++){
                                        outbuf[outbufindex][i]=(Int16)(((long)y[2*i]*raisedhanning[i])>>15);
                                }

                                bufstore[qq] = bindex;
                                qq++;
                                qq = qq>=lqq?0:qq;
                }

        }               //end while loop

        /*
        EZDSP5502_MCBSP_close(); // Disable McBSP
        AIC3204_rset( 1, 1 );   // Reset codec
        */

        return 0;
}


Int16 readswitches(Int16 *switches){
        /* Detect Switches */
        switches[0]=!(EZDSP5502_I2CGPIO_readLine(SW0));
        switches[1]=!(EZDSP5502_I2CGPIO_readLine(SW1));

        return 0;
}

/*build wHat andhopaXwHat vectors (radians normalized
 * to [-1,1] range in Q.15 format)*/
Int16 build_wHatVecs(DATA *wHat,float *fwHat,DATA *hopaXwHat)
{
        Int16 k;
        for(k=0;k<NFFT/2;k++){
                wHat[k] = (Int16)((long)k*(0x7FFF/(NFFT/2)));
                hopaXwHat[k] = wHat[k]*HOPA;
                fwHat[k] = (2.0f*k)/NFFT;
        }
        for(k=NFFT-1;k>NFFT/2;k--){
                wHat[k] = -wHat[NFFT-k];
                hopaXwHat[k] = -hopaXwHat[NFFT-k];
                fwHat[k]=-1.0f*fwHat[NFFT-k];
        }
                wHat[NFFT/2] = 0x8000;
                hopaXwHat[k] = -HOPA;
                fwHat[k] = -1.0;
        return 0;
}

Int16 build_Windows(DATA *hanning, DATA *raisedhanning){
        Int16 k;
        float temp;
        for(k=0;k<NFFT;k++){
                temp = 0.5*(1-cos(6.2831858*k/(NFFT-1)));
                fltoq15(&temp,&hanning[k],1);
                temp = pow(temp,HPOW);
```

```
                        fltoq15(&temp,&raisedhanning[k],1);
                }
                return 0;
}

void config_AIC3204()
{

   /* ---------------------------------------------------------------- *
    *  Configure AIC3204                                              *
    * ---------------------------------------------------------------- */
   AIC3204_rset( 0, 0 );      // Select page 0
   AIC3204_rset( 1, 1 );      // Reset codec
   AIC3204_rset( 0, 1 );      // Select page 1
   AIC3204_rset( 1, 8 );      // Disable crude AVDD generation from DVDD
   AIC3204_rset( 2, 1 );      // Enable Analog Blocks, use LDO power
   AIC3204_rset( 0, 0 );

   /* PLL and Clocks config and Power Up  */
   AIC3204_rset( 27, 0x0d );  // BCLK and WCLK are set as o/p; AIC3204(Master)
   AIC3204_rset( 28, 0x00 );  // Data ofset = 0
   AIC3204_rset( 4, 3 );      // PLL setting: PLLCLK <- MCLK, CODEC_CLKIN <-PLL CLK
   AIC3204_rset( 6, 7 );      // PLL setting: J=7
   AIC3204_rset( 7, 0x06 );   // PLL setting: HI_BYTE(D=1680)
   AIC3204_rset( 8, 0x90 );   // PLL setting: LO_BYTE(D=1680)
   AIC3204_rset( 30, 0x9C );  // For 32 bit clocks per frame in Master mode ONLY
                       // BCLK=DAC_CLK/N =(12288000/8) = 1.536MHz = 32*fs
   AIC3204_rset( 5, 0x91 );   // PLL setting: Power up PLL, P=1 and R=1
   AIC3204_rset( 13, 0 );     // Hi_Byte(DOSR) for DOSR = 128 decimal or 0x0080 DAC oversamppling
   AIC3204_rset( 14, 0x80 );  // Lo_Byte(DOSR) for DOSR = 128 decimal or 0x0080
   AIC3204_rset( 20, 0x80 );  // AOSR for AOSR = 128 decimal or 0x0080 for decimation filters 1 to 6
   AIC3204_rset( 11, 0x82 );  // Power up NDAC and set NDAC value to 2
   AIC3204_rset( 12, 0x87 );  // Power up MDAC and set MDAC value to 7
   AIC3204_rset( 18, 0x87 );  // Power up NADC and set NADC value to 7
   AIC3204_rset( 19, 0x82 );  // Power up MADC and set MADC value to 2

   /* DAC ROUTING and Power Up */
   AIC3204_rset( 0, 1 );      // Select page 1
   AIC3204_rset( 0x0c, 8 );   // LDAC AFIR routed to HPL
   AIC3204_rset( 0x0d, 8 );   // RDAC AFIR routed to HPR
   AIC3204_rset( 0, 0 );      // Select page 0
   AIC3204_rset( 64, 2 );     // Left vol=right vol
   AIC3204_rset( 65, 0);      // Left DAC gain to 0dB VOL; Right tracks Left
   AIC3204_rset( 63, 0xd4 );  // Power up left,right data paths and set channel
   AIC3204_rset( 0, 1 );      // Select page 1
   AIC3204_rset( 9, 0x30 );   // Power up HPL,HPR
   AIC3204_rset( 0x10, 0x00 );// Unmute HPL , 0dB gain
   AIC3204_rset( 0x11, 0x00 );// Unmute HPR , 0dB gain
   AIC3204_rset( 0, 0 );      // Select page 0
   EZDSP5502_waitusec( 1000000 ); // wait

   /* ADC ROUTING and Power Up */
   AIC3204_rset( 0, 1 );      // Select page 1
   AIC3204_rset( 0x34, 0x30 );// STEREO 1 Jack
                       // IN2_L to LADC_P through 40 kohm
   AIC3204_rset( 0x37, 0x30 );// IN2_R to RADC_P through 40 kohmm
   AIC3204_rset( 0x36, 3 );   // CM_1 (common mode) to LADC_M through 40 kohm
   AIC3204_rset( 0x39, 0xc0 );// CM_1 (common mode) to RADC_M through 40 kohm
   AIC3204_rset( 0x3b, 0 );   // MIC_PGA_L unmute
   AIC3204_rset( 0x3c, 0 );   // MIC_PGA_R unmute
   AIC3204_rset( 0, 0 );      // Select page 0
   AIC3204_rset( 0x51, 0xc0 );// Powerup Left and Right ADC
   AIC3204_rset( 0x52, 0 );   // Unmute Left and Right ADC
   AIC3204_rset( 0, 0 );
   EZDSP5502_waitusec( 200 ); // Wait

             return;
}
```

ezdsp5502_mcbsp.c

This is the portion of code that contains the interrupt handler that accepts input from and sends output to the codec.

```
#include "ezdsp5502_mcbsp.h"
#include "csl_mcbsp.h"
#include "stdio.h"
#include "csl_irq.h"
#include "math.h"
#include "dsplib.h"
#include "pvplaydefs.h"

extern void VECSTART(void);
extern Int16 inbuf[NFFT];
extern Int16 outbuf[NOUT][NFFT];
extern Int16 bufindex;
extern Int16 outbufindex;
extern Int16 Px;
extern Int16 hops_[NPHASE];
extern Int16 fr_[NPHASE];

Int16 p,k;
Int16 dataval;
Int16 intrcnt;
Int16 outindex;
Int16 outsum;
Int16 outterm;
Int16 tmp;
long int avg=0;
Int16 prevavg=0;
Int16 a[FILTN] = FILTA;
Int16 b[FILTN] = FILTB;
Int16 butterbufout[FILTN];
Int16 butterbufin[FILTN];
Int16 prefiltin[FILTN];
Int16 prefiltout[FILTN];
long int ltemp=0;
#pragma DATA_SECTION (ftemp,".f4")
float ftemp=0;

#pragma DATA_SECTION (outcapture,".miscCE0");
Int16 outcapture[2*NFFT];
#pragma DATA_SECTION (outcapture2,".miscCE0");
Int16 outcapture2[2*NFFT];
#pragma DATA_SECTION (outcapture3,".miscCE0");
Int16 outcapture3[2*NFFT];

interrupt void readISR(void);

MCBSP_Handle aicMcbsp;

/*
 *
 *  EZDSP5502_MCBSP_init( )
 *
 *  Description
 *      Enable and initalize the MCBSP module for use
 *      with the AIC3204.
 *
 */
Int16 EZDSP5502_MCBSP_init( )
{
   /* Set values for MCBSP configuration structure */
   MCBSP_Config ConfigLoopBack32= {
       MCBSP_SPCR1_RMK(
        MCBSP_SPCR1_DLB_OFF,          // DLB    = 0
        MCBSP_SPCR1_RJUST_RZF,        // RJUST  = 0 (Right Justified)
        MCBSP_SPCR1_CLKSTP_DISABLE,   // CLKSTP = 0
        MCBSP_SPCR1_DXENA_NA,         // DXENA  = 0
        MCBSP_SPCR1_ABIS_DISABLE,     // ABIS   = 0
        MCBSP_SPCR1_RINTM_RRDY,       // RINTM  = 0
        MCBSP_SPCR1_RSYNCERR_NO ,     // RSYNCERR = 0
        MCBSP_SPCR1_RRST_DISABLE      // RRST   = 0
        ),
      MCBSP_SPCR2_RMK(
        MCBSP_SPCR2_FREE_NO,          // FREE   = 1 (Free running)
        MCBSP_SPCR2_SOFT_NO,          // SOFT   = 0
        MCBSP_SPCR2_FRST_RESET,       // FRST   = 0 (External FS)
        MCBSP_SPCR2_GRST_RESET,       // GRST   = 0 (All Clocks External)
        MCBSP_SPCR2_XINTM_XRDY,       // XINTM  = 0
        MCBSP_SPCR2_XSYNCERR_NO,      // XSYNCERR = N/A
        MCBSP_SPCR2_XRST_DISABLE      // XRST   = 0
        ),
      MCBSP_RCR1_RMK(
        MCBSP_RCR1_RFRLEN1_OF(1),     // RFRLEN1 = 1 (2 Words)
        MCBSP_RCR1_RWDLEN1_16BIT      // RWDLEN1 = 2 (16-bit)
        ),
      MCBSP_RCR2_RMK(
        MCBSP_RCR2_RPHASE_SINGLE,     // RPHASE  = 0 (Single Phase)
        MCBSP_RCR2_RFRLEN2_OF(0),     // RFRLEN2 = 0
        MCBSP_RCR2_RWDLEN2_8BIT,      // RWDLEN2 = 0
        MCBSP_RCR2_RCOMPAND_MSB,      // RCOMPAND = 0
        MCBSP_RCR2_RFIG_YES,          // RFIG    = 0
        MCBSP_RCR2_RDATDLY_1BIT       // RDATDLY = 11 (1-bit delay)
        ),
      MCBSP_XCR1_RMK(
        MCBSP_XCR1_XFRLEN1_OF(1),     // XFRLEN1 = 1 (2 Words)
        MCBSP_XCR1_XWDLEN1_16BIT      // XWDLEN1 = 2 (16-bit)
        ),
      MCBSP_XCR2_RMK(
        MCBSP_XCR2_XPHASE_SINGLE,     // XPHASE  = 0 (Single Phase)
        MCBSP_XCR2_XFRLEN2_OF(0),     // XFRLEN2 = 0
        MCBSP_XCR2_XWDLEN2_8BIT,      // XWDLEN2 = 0
        MCBSP_XCR2_XCOMPAND_MSB,      // XCOMPAND= 0
        MCBSP_XCR2_XFIG_YES,          // XFIG    = 0
        MCBSP_RCR2_RDATDLY_1BIT       // XDATDLY = 1 (1-bit delay)
        ),
      MCBSP_SRGR1_RMK(
        MCBSP_SRGR1_FWID_OF(0),       // FWID    = 0 (All Clocks External)
        MCBSP_SRGR1_CLKGDV_OF(0)      // CLKGDV  = 0 (All Clocks External)
        ),
      MCBSP_SRGR2_RMK(
        MCBSP_SRGR2_GSYNC_FREE,       // FREE    = 0
        MCBSP_SRGR2_CLKSP_RISING,     // CLKSP   = 0
        MCBSP_SRGR2_CLKSM_CLKS,       // CLKSM   = 0
        MCBSP_SRGR2_FSGM_DXR2XSR,    // FSGM    = 0
        MCBSP_SRGR2_FPER_OF(0)       // FPER    = 0
        ),
      MCBSP_MCR1_DEFAULT,
      MCBSP_MCR2_DEFAULT,
      MCBSP_PCR_RMK(
        MCBSP_PCR_XIOEN_SP,          /* XIOEN    = 0   */
        MCBSP_PCR_RIOEN_SP,          /* RIOEN    = 0   */
        MCBSP_PCR_FSXM_EXTERNAL,     /* FSXM     = 0   */
        MCBSP_PCR_FSRM_EXTERNAL,     /* FSRM     = 0   */
        MCBSP_PCR_CLKXM_INPUT,       /* CLKXM    = 0   */
        MCBSP_PCR_CLKRM_INPUT,       /* CLKRM    = 0   */
        MCBSP_PCR_SCLKME_NO,         /* SCLKME   = 0   */
        0,                           /* DXSTAT = N/A   */
        MCBSP_PCR_FSXP_ACTIVEHIGH,   /* FSXP     = 0   */
        MCBSP_PCR_FSRP_ACTIVELOW,    /* FSRP     = 0   */
        MCBSP_PCR_CLKXP_FALLING,     /* CLKXP    = 1   */
        MCBSP_PCR_CLKRP_FALLING      /* CLKRP    = 1   */
        ),
      MCBSP_RCERA_DEFAULT,
      MCBSP_RCERB_DEFAULT,
      MCBSP_RCERC_DEFAULT,
      MCBSP_RCERD_DEFAULT,
      MCBSP_RCERE_DEFAULT,
      MCBSP_RCERF_DEFAULT,
      MCBSP_RCERG_DEFAULT,
      MCBSP_RCERH_DEFAULT,
      MCBSP_XCERA_DEFAULT,
      MCBSP_XCERB_DEFAULT,
      MCBSP_XCERC_DEFAULT,
      MCBSP_XCERD_DEFAULT,
      MCBSP_XCERE_DEFAULT,
      MCBSP_XCERF_DEFAULT,
      MCBSP_XCERG_DEFAULT,
      MCBSP_XCERH_DEFAULT
      };

      Uint16 rcvEventID;

  /* Close any previous instance of the McBSP */
  EZDSP5502_MCBSP_close( );

  /* Open McBSP port and associate with handle */
  aicMcbsp = MCBSP_open(MCBSP_PORT1, MCBSP_OPEN_RESET);

          /*Interrupt setup*/
          IRQ_globalDisable();
          rcvEventID = MCBSP_getRcvEventId(aicMcbsp);
          IRQ_setVecs((Uint32)(&VECSTART));
  /* Clear any pending receive or transmit interrupts */
  IRQ_clear(rcvEventID);
  IRQ_plug(rcvEventID, &readISR);


  /* Configure McBSP with values */
  MCBSP_config(aicMcbsp, &ConfigLoopBack32);


  IRQ_enable(rcvEventID);
  /* Enable MCBSP transmit and receive */
  MCBSP_start(aicMcbsp,MCBSP_RCV_START | MCBSP_XMIT_START,0 );
  EZDSP5502_waitusec( 42 ); // Wait 2 sample rate periods
```

```
		IRQ_globalEnable();
		printf("aic setup complete...\n");
   return 0;
}

/*
 *
 *  EZDSP5502_MCBSP_close( )
 *
 *  Description
 *      Closes MCBSP associated with handle
 *
 */
Int16 EZDSP5502_MCBSP_close( )
{
   MCBSP_close(aicMcbsp);

   return 0;
}

/*
 *
 *  EZDSP5502_MCBSP_write( data )
 *
 *      Int16 data   <-  16-bit audio data
 *
 *  Description
 *      Sends data out the McBSP
 *
 */
void EZDSP5502_MCBSP_write( Int16 data )
{
   while(!MCBSP_xrdy(aicMcbsp));
   MCBSP_write16(aicMcbsp, data);
}

/*
 *
 *  EZDSP5502_MCBSP_read( *data )
 *
 *      Int16* data   <-  Pointer to location to store received data
 *
 *  Description
 *      Copies data received by the McBSP to the location of the pointer
 *
 */
void EZDSP5502_MCBSP_read( Int16* data )
{
   while(!MCBSP_rrdy(aicMcbsp));
   *data = MCBSP_read16(aicMcbsp);
}


interrupt void readISR(void){

		if(MCBSP_rrdy(aicMcbsp)){
			intrcnt++;
			dataval = MCBSP_read16(aicMcbsp);

			if((intrcnt & 0x7)==0){
```

```
		//outcapture[bufindex] = dataval;///////////DEBUG
		dataval = (dataval&INMASK)>>INRSHIFT;
		avg += dataval;
		dataval -= prevavg;
		//filter
		prefiltin[0] = dataval;
		ltemp=(((long)prefiltin[0]*b[0])>>FILTFRM);
		for(p=1;p<FILTN;p++){
			ltemp+=(((long)prefiltin[p]*b[p])>>FILTFRM);
			ltemp-=(((long)prefiltout[p]*a[p])>>FILTFRM);
		}
		prefiltout[0] = (Int16)ltemp;
		for(p=0;p<FILTN-1;p++){
			prefiltin[FILTN-1-p] = prefiltin[FILTN-2-p];
			prefiltout[FILTN-1-p] = prefiltout[FILTN-2-p];
		}
		dataval = prefiltout[0];
		inbuf[bufindex] = dataval;
		outcapture[bufindex] = inbuf[bufindex];///////////DEBUG
		tmp = bufindex;
		while(tmp>=HOPA){
			tmp -= HOPA;
		}
		outsum=0;
		for(p=0;p<NOUT-1;p++){
			k = outbufindex-1-p;
			k = k<0?NOUT+k:k;
			outindex = p*hops_[Px]+(Int16)(((long)tmp*fr_[Px])>>14);
			if(outindex>=0 && outindex<NFFT){
				outsum = outsum + (outbuf[k][outindex]>>OUTSUMRSHIFT);
			}
		}

		outterm = outsum;
		outcapture2[bufindex] = outterm;///////////DEBUG
		//filter
		butterbufin[0] = outterm;
		ltemp=(((long)butterbufin[0]*b[0])>>FILTFRM);
		for(p=1;p<FILTN;p++){
			ltemp+=(((long)butterbufin[p]*b[p])>>FILTFRM);
			ltemp-=(((long)butterbufout[p]*a[p])>>FILTFRM);
		}
		butterbufout[0] = (Int16)ltemp;
		for(p=0;p<FILTN-1;p++){
			butterbufin[FILTN-1-p] = butterbufin[FILTN-2-p];
			butterbufout[FILTN-1-p] = butterbufout[FILTN-2-p];
		}

		butterbufout[0] =(butterbufout[0]<<OUTLSHIFT)+OUTDC;
		MCBSP_write16(aicMcbsp,butterbufout[0]);

		outcapture3[bufindex]=butterbufout[0];/////DEBUG
		bufindex++;
		if(bufindex==2*NFFT){
			bufindex=0;
			prevavg=avg/(2*NFFT)+AVGOFFSET;
			avg=0;
		}
	}
	/* repeat output (mono) */
	else{
		MCBSP_write16(aicMcbsp,butterbufout[0]);
```

```
                    }
            }
}
```

pvplaydefs.h

This header file contains definitions of various algorithm parameters.

```
#ifndef PVPLAYDEFS_H_
#define PVPLAYDEFS_H_
#define NFFT        256         //number of fft samples to process at a time
#define HOPA        64          //Don't change this one (NFFT/4)
#define INMASK 0xFFFC
#define INRSHIFT 0
#define INRSHIFT2 2
#define NPHASE   3
#define FR                      {0x3800,0x4000,0x4800}     //in 2.14
#define HOPS        {56,64,72}            //HOPA*FR, round to int
//#define FR                    {0x3C00,0x4000,0x4400}     //in 2.14
//#define HOPS      {60,64,68}            //HOPA*FR, round to int
#define NOUT       5
#define OUTSUMRSHIFT 2
#define AVGOFFSET 0
#define OUTDC 0//0x4000
#define OUTLSHIFT 0
#define HPOW 1.00f//1.25f

//output filter - 2nd order Butterworth lowpass, 2.14 format
#define FILTN 3
#define FILTFRM 15
#define FILTB {12254,24508,12254}
#define FILTA {32767,10078,6172}

#endif /*PVPLAYDEFS_H_*/
```

lnkx.cmd

This is the linker file for the build, which tells the linker where code segments will reside in the memory.

```
/*-stack    0x2000      /* Primary stack size   */
-stack    0x1200
/*-sysstack 0x1000*/
-sysstack 0x0D00      /* Secondary stack size */
/*-heap     0x2000 */      /* Heap area size       */
-heap         0x1000
-c                  /* Use C linking conventions: auto-init vars at runtime */
-u _Reset            /* Force load of reset interrupt handler              */

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
 PAGE 0:  /* ---- Unified Program/Data Address Space ---- */

  MMR    (RWIX): origin = 0x000000, length = 0x0000C0  /* MMRs */
  BTRSVD (RWIX): origin = 0x0000C0, length = 0x000240  /* Reserved for Boot Loader */
  DARAM  (RWIX): origin = 0x000300, length = 0x00FB00  /* 64KB - MMRs - VECS*/
  VECS   (RWIX): origin = 0x00FE00, length = 0x000200  /* 256 bytes Vector Table */
  CE0         : origin = 0x010000, length = 0x3f0000  /* 4M minus 64K Bytes */
  CE1         : origin = 0x400000, length = 0x400000
 CE2         : origin = 0x800000, length = 0x400000
 CE3         : origin = 0xC00000, length = 0x3F8000
 PDROM   (RIX): origin = 0xFF8000, length = 0x008000  /*  32KB */


PAGE 2:  /* -------- 64K-word I/O Address Space -------- */

 IOPORT (RWI) : origin = 0x000000, length = 0x020000
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   /* Here's where the program should go */
   .text     >  DARAM align(32) fill = 20h { * (.text)  }

   /* Primary system stack        */
   .stack    >  DARAM align(32) fill = 00h
   /* Secondary system stack      */
   .sysstack >  DARAM align(32) fill = 00h
   /* CSL data                    */
   .csldata  >  DARAM align(32) fill = 00h
   /* Initialized vars            */
   .data     >  DARAM align(32) fill = 00h
    /* Global & static vars        */
   .bss      >  DARAM align(32) fill = 00h
   /* Constant data               */
   .const    >  DARAM align(32) fill = 00h
   /* Dynamic memory (malloc)     */
   .sysmem   >  DARAM
   /* Switch statement tables     */
   .switch   >  DARAM
   /* Auto-initialization tables  */
   .cinit    >  DARAM
   /* Initialization fn tables    */
   .pinit    >  DARAM align(32) fill = 00h
   /* C I/O buffers               */
   .cio      >  DARAM align(32) fill = 00h
    /* Arguments to main()         */
   .args     >  DARAM align(32) fill = 00h

   /* Used to store FFT arrays */
   .input    >  DARAM align(32)

             /* interrupt vector table must be on 256 "page" boundry*/
             .vectors  >  VECS
             vectors    >            VECS

   .ioport   >  IOPORT PAGE 2         /* Global & static ioport vars */

   /* More FFT items */
   .fftcode : {} > DARAM PAGE 0

   .input   : {} > DARAM PAGE 0, align(32)  /* this is due to long-word data memory access */

   /* for atan2 */
   .data     : {} > DARAM PAGE 0

   /* LUTs */
   .LUTs   : {} > CE0 PAGE 0
```

```
    /* FFT Twiddle factors - length 0x400 */
    .twiddle : {} > CE0 PAGE 0, align(2048)

    /* float data that uses Q15 conversion functions */
    .f1              : {} > CE0 PAGE 0, align(32)
    .f2              : {} > CE0 PAGE 0, align(32)
    .f3              : {} > CE0 PAGE 0, align(32)
    .f4              : {} > CE0 PAGE 0, align(32)

    .miscCE0 : {} > CE0 PAGE 0

    .test   : {} > CE0 PAGE 0

}
```